

Enforcing Operational Properties including Blockfreeness for Deterministic Pushdown Automata

S. Schneider and U. Nestmann

Technische Universität Berlin

Abstract: We present an algorithm which modifies a deterministic pushdown automaton (DPDA) such that (i) the marked language is preserved, (ii) lifelocks are removed, (iii) deadlocks are removed, (iv) all states and edges are accessible, and (v) operational blockfreeness is established (i.e., coaccessibility in the sense that every initial derivation can be continued to a marking configuration). This problem can be trivially solved for deterministic finite automata (DFA) but is not solvable for standard petri net classes. The algorithm is required for an operational extension of the supervisory control problem (SCP) to the situation where the specification is modeled by a DPDA.

Keywords: Deterministic Pushdown Automata, DPDA, Blockingness, Deadlocks, Lifelocks, Accessibility, Coaccessibility, Supervisory Control

We are introducing an algorithm to transform a DPDA such that its observable operational behavior is restricted to its desired fragment. The algorithm decomposes the problem into three steps: transformation of the DPDA into a Context Free Grammar (CFG) while preserving the operational behavior, restricting the CFG to enforce operational blockfreeness, and the transformation of the resulting CFG via Parsers to DPDA while preserving and establishing the relevant criteria on the operational behavior. The algorithm presented here is an essential part for the effective solution of the supervisory control problem for DFA plants and DPDA specifications which is reduced (in the companion paper by Schneider, Schmuck, Raisch, and Nestmann (2014)) to the effective implementability of ensuring blockfreeness (solved in this paper) and ensuring controllability (solved in the companion paper by Schmuck, Schneider, Raisch, and Nestmann (2014)).

In Section 1 we define abstract transition systems (ATS) as a basis for the systems involved in the algorithm and give a formal problem statement to be solved for DPDA. In Section 2 we define the concrete transition systems appearing in the algorithm as instantiations of ATS. In Section 3 we present the extensive algorithm due to space restrictions mostly informally using a running example before we discuss the formal verification and possible improvements of the approach. The formal constructions of the algorithm are contained in Schneider and Schmuck (2013). We summarize our results in Section 4 and outline our next steps in Section 5.

1. ABSTRACT TRANSITION SYSTEMS

The concrete systems used in this paper (including DPDA, CFG, and Parsers) are instantiations of the subsequently defined class of Abstract Transition Systems (ATS). Thus, they will inherit the uniform definitions of derivations,

languages, and the problem to be solved from the ATS definitions.

Throughout the paper we use the following notations.

Notation 1. Let A be an alphabet and let B be a set. Then (i) A^* denotes the set of all finite words over A , (ii) $A^{\leq 1} = A \cup \{\lambda\}$, (iii) $A^{\omega*}$ denotes the set of all finite and infinite words over A , (iv) \cdot is the (sometimes omitted) concatenation operation on words (and languages), (v) \sqsubseteq is the prefix relation, (vi) \bar{A} is the prefix-closure of A , (vii) \supseteq is the suffix relation, and (viii) $k:w$ denotes the k -Prefix of $w \in A^*$ defined by (if $w = \alpha \cdot w' \wedge k > 0$ then $\alpha \cdot ((k-1):w')$ else λ), and (ix) $\boxtimes(A, B)$ denotes $(A \cup \{\perp\}) \times B$ where \perp represents undefinedness. \square

Definition 1. (Abstract Transition System).

$\mathcal{S} = (E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um}) \in \text{ATS}$ iff (i) E is a set of step-edges, (ii) C is a set of configurations, (iii) S is a set of states, (iv) π_S maps each configuration to at most one state, (v) R is a binary step-relation on $\boxtimes(E, C)$, (vi) $c_0 \in C$ is the initial configuration, (vii) A is the marking subset of C , (viii) O is the set of outputs, and (ix) $o_{um} : C \rightarrow 2^O$ and $o_m : A \rightarrow 2^O$ define the unmarked and marked outputs for configurations. \square

For these ATS we define their derivations, generated languages, and subsequently the properties to be enforced.

Definition 2. (Semantics of ATS). (i) the set of derivations $\mathcal{D}(\mathcal{S})$ contains all elements from $\boxtimes(E, C)^{\omega*}$ starting in a configuration of the form (\perp, c) where all adjacent $(c_1, e_1), (c_2, e_2) \in \boxtimes(E, C)$ satisfy $(c_1, e_1) R (c_2, e_2)$, (ii) the set of initial derivations $\mathcal{D}_I(\mathcal{S})$ contains all elements of $\mathcal{D}(\mathcal{S})$ starting with (\perp, c_0) , (iii) the reachable configurations $\mathcal{C}_{\text{reach}}(\mathcal{S})$ are defined by $\{c \in C \mid \exists d \in \mathcal{D}_I(\mathcal{S}) \cdot d(n) = (e, c)\}$, (iv) the marked language $L_m(\mathcal{S})$ is defined by $\cup o_m(F \cap \mathcal{C}_{\text{reach}}(\mathcal{S}))$, and (v) the unmarked language $L_{um}(\mathcal{S})$ is defined by $\cup o_{um}(\mathcal{C}_{\text{reach}}(\mathcal{S}))$.

The concatenation of derivations $d_1, d_2 \in \mathcal{D}(\mathcal{S})$ is given by $(d_1 \cdot_n d_2)(i) = (\text{if } i \leq n \text{ then } d_1(i) \text{ else } d_2(i - n))$. \square

Definition 3. (Properties of ATS). (i) \mathcal{S} has a deadlock iff for some finite $d \in \mathcal{D}_1(\mathcal{S})$ of length $n \in \mathbb{N}$ which is not marking (i.e., for all k , $d(k) = (e, c)$ implies $c \notin A$) there is no c' such that $d(n) R c'$, (ii) \mathcal{S} has a lifelock iff for some infinite $d \in \mathcal{D}_1(\mathcal{S})$ there is an $N \in \mathbb{N}$ such that the unmarked language of d is constant after N (i.e., for all $k \geq N$, $o_{um}(d(N)) = o_{um}(d(k))$), (iii) \mathcal{S} is accessible iff for each $p \in S$ there is $c \in \mathcal{C}_{reach}(\mathcal{S})$ such that $\pi_S(c) = p$ and for each $e \in E$ there is $d \in \mathcal{D}_1(\mathcal{S})$ such that $d(n) = (e, c)$, and (iv) \mathcal{S} is operational blockfree iff for any finite $d_i \in \mathcal{D}_1(\mathcal{S})$ of length $n \in \mathbb{N}$ ending in $d_i(n) = (e, c)$ there is a continuation $d_c \in \mathcal{D}(\mathcal{S})$ such that $d_i \cdot_n d_c$ is a marking derivation and d_i and d_c match at the gluing point n (i.e., $d_c(0) = (\perp, c)$). \square

By definition, for operational blockfree ATS the absence of deadlocks is guaranteed. Finally, we present the problem of enforcing the desired properties on an ATS, which will be solved for DPDA by the algorithm presented in Section 3.

Definition 4. (Problem Statement for ATS). Let $\mathcal{S} \in \text{ATS}$. How to find $\mathcal{S}' \in \text{ATS}$ such that (i) $L_m(\mathcal{S}) = L_m(\mathcal{S}')$, (ii) \mathcal{S}' is accessible, (iii) \mathcal{S}' has no deadlocks, (iv) \mathcal{S}' has no lifelocks, and (v) \mathcal{S}' is operational blockfree? \square

In the DFA-setting: lifelocks can not occur and the other aspects of the problem are solved by simple and efficient graph-traversal algorithms pruning out states which are either not reachable from the initial state or from which no marking state can be reached¹.

2. CONCRETE TRANSITION SYSTEMS

Every deterministic context free language can be properly represented by at least three different types of finite models: a deterministic EPDA, a context free grammar (CFG) satisfying the LR(1) determinism property, and a deterministic Parser. These three types occur at intermediate steps of our algorithm which solves the problem stated in Definition 4. Therefore, the following subsections contain their definitions as instantiations of the ATS. In each of the three cases we proceed in three steps: (1) definition of EPDA, CFG, and Parser as tuples, (2) instantiation of the ATS-scheme by defining each of the ten components, and (3) characterization of the determinism conditions.

Remark 1. We provide the slightly nonstandard branching semantics² for EPDA and Parsers which utilize a history variable in the configurations to greatly simplify the definition of the operational-blockfreeness from Definition 3. Furthermore, this branching semantics corresponds to the intuition that the finite state realizations are generators rather than acceptors of languages, as it is customary in the context of supervisory control theory. \square

2.1 EPDA and DPDA

We introduce EPDA, which are NFA enriched with a variable on which the stack-operations top, pop, and, push can be executed.

¹ The trivial handling of an ATS with empty marked language obtained at some point of the calculation is kept implicit in this paper (in this case, no solution exists and the calculation can be aborted).

² The branching interpretation is already the standard for CFG.

	EPDA	PDA	DPDA	NFA	DFA
1-popping		✓	✓	✓	✓
deterministic			✓		✓
λ -step-free				✓	✓
stack-free				✓	✓

Table 1. Subclasses of EPDA.

Definition 5. (Extended Pushdown Automata (EPDA)). $M = (Q, \Sigma, \Gamma, \delta, p_0, \square, F) \in \text{EPDA}$ iff (i) the states Q , the output alphabet Σ , the stack alphabet Γ , and the set of edges δ are finite ($Q, Q^*, \Sigma, \Sigma^*, \Gamma, \Gamma^*$ range over $p, \tilde{p}, \alpha, w, \gamma, s$, respectively), (ii) $\delta : Q \times \Sigma^{\leq 1} \times \Gamma^* \times \Gamma^* \times Q$, (iii) the end-of-stack marker \square is contained in Γ , (iv) the marking states F and the initial state p_0 are contained in Q , and (v) \square is never removed from the stack (i.e., $(p, \sigma, s, s', p') \in \delta$ and $s \sqsupseteq \square$ imply $s' \sqsupseteq \square$). \square

We proceed with the ATS instantiation for EPDA.

Definition 6. (EPDA—ATS Instantiation).

An EPDA $M = (Q, \Sigma, \Gamma, \delta, p_0, \square, F)$ instantiates the ATS scheme $(E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um})$ via: (i) $\overline{E} \delta$ (ii) $\overline{C} \mathcal{C}(M) \triangleq Q \times \Sigma^* \times \Gamma^*$ where $(p, w, s) \in \mathcal{C}(M)$ consists of a state p , a history variable w (storing the symbols generated), and the stack-variable s (iii) $\overline{S} Q$ (iv) $\overline{\pi_S}(p, w, s) p$ (v) $\overline{R} \vdash_M : \mathfrak{X}(\delta, \mathcal{C}(M))^2$ defined by $(e, (p, w, s', s)) \vdash_M ((p, \sigma, s', s''), p', w \cdot \sigma, s'' \cdot s)$ (vi) $\overline{c_0} (p_0, \lambda, \square)$ (vii) $\overline{A} \{(p, w, s) \in \mathcal{C}(M) \mid p \in F\}$ (viii) $\overline{O} \Sigma^*$ (ix) $\overline{o_m}(p, w, s, o_{um}(p, w, s)) \{w\}$ \square

The well known sub-classes of EPDA having one or more of the properties below are defined in Table 1.

Definition 7. (Sub-classes of EPDA). An EPDA is 1-popping iff every edge pops precisely one element from Γ from the stack. An EPDA is deterministic iff for every reachable configuration all two distinct steps append distinct elements of Σ to the history variable³. An EPDA is λ -step-free iff no edge is of the form (p, λ, s, s', p') . An EPDA is stack-free iff every edge is of the form $(p, \alpha, \square, \square, p')$. \square

2.2 CFG and LR(1)

A CFG (e.g., defined by Ginsburg and Greibach (1966)) is a term-replacement system replacing a nonterminal with a word over output symbols⁴ and nonterminals.

Definition 8. (Context-Free Grammars (CFG)).

$G = (N, \Sigma, P, S) \in \text{CFG}$ iff (i) the nonterminals N (ranging over A, B), the output alphabet Σ , and the productions P are finite (ii) $P : N \times (N \cup \Sigma)^*$, and (iii) the initial nonterminal S is contained in N . $N \cup \Sigma$ and $(N \cup \Sigma)^*$ range over κ and v , respectively. Productions (A, v) are written $A \rightarrow v$. \square

Definition 9. (CFG—ATS Instantiation).

A CFG $G = (N, \Sigma, P, S)$ instantiates the ATS scheme $(E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um})$ via: (i) $\overline{E} P$ (ii) $\overline{C} \mathcal{C}(G) = (N \cup \Sigma)^*$ (iii) $\overline{S} N$ (iv) $\overline{\pi_S}$ take the first nonterminal (if present) of the configuration (v) $\overline{R} \vdash_G : \mathfrak{X}(P, \mathcal{C}(G))^2$ given by $(e, (v_1 \cdot A \cdot v_2)) \vdash_G ((A, v), v_1 \cdot v \cdot v_2)$ (vi) $\overline{c_0} S$ (vii) $\overline{A} \Sigma^*$ (viii) $\overline{O} \Sigma^*$ (ix) $\overline{o_m}(v) \{v\}$ (x) $\overline{o_{um}}(v) \{v\}$ \square

³ Thus, λ -steps may not be enabled simultaneously with other steps.

⁴ The output symbols of a CFG are usually called terminals.

The LR(1)-condition below, which corresponds to the determinism property of EPDA, depends on the restriction of the step-relation to the replacement of the right-most nonterminal which will be denoted by the index *rm*.

Definition 10. (LR(1)-Condition). According to Sippu and Soisalon-Soininen (1990) (page 52)⁵, LR(1) is the set of all CFG for which (assuming $x \in \Sigma^*$)

- (i) $(\perp, S) \vdash_G^{\text{rm}*} (e_1, v'_1 \cdot A_1 \cdot w_1) \vdash_G^{\text{rm}} ((A_1, v_1), v'_1 \cdot v_1 \cdot w_1)$,
- (ii) $(\perp, S) \vdash_G^{\text{rm}*} (e_2, v'_2 \cdot A_2 \cdot w_2) \vdash_G^{\text{rm}} ((A_2, v_2), v'_2 \cdot v_2 \cdot w_2)$,
- (iii) $v'_2 \cdot v_2 = v'_1 \cdot v_1 \cdot x$, and
- (iv) $1:w_1 = 1:(x \cdot w_2)$, imply
- (v) $v'_1 = v'_2$, $A_1 = A_2$, and $v_1 = v_2$. \square

Intuitively, if a parser for a CFG has generated the shorter prefix $v'_1 \cdot v_1$ it must be able to decide by fixing the next symbol ($1:w_1$ and $1:(x \cdot w_2)$, respectively) whether (A_1, v_1) is to be applied backwards or whether for $x \neq \lambda$ another symbol of x should be generated or for $x = \lambda$ the production (A_2, v_2) is to be applied backwards⁶.

2.3 Parser

Intuitively, a Parser is an EPDA with mild modifications⁷: (1) the parser may fix the next output-symbol (without generating it) and (2) the parser may terminate the generation of symbols (by fixing the end-of-output marker \diamond).

Definition 11. (Parser). $M = (N, \Sigma, S, F, P, \diamond) \in \text{Parser}$ iff (i) the stack alphabet N , the output alphabet Σ , the marking stack-tops F , and the rules P are finite, $(N, N^*, \Sigma, \Sigma^*, \text{range over } p, \bar{p}, \alpha, w, \text{ respectively})$ (ii) $P : N^+ \times \Sigma^{\leq 1} \times N^+ \times \Sigma^{\leq 1}$, (iii) the initial stack symbol S and the marking stack-tops F are contained in N , (iv) the end-of-output marker \diamond is contained in Σ , (v) the parser may not modify the output (i.e., $(s \cdot p, w, s' \cdot p', w') \in P$ implies $w \sqsupseteq w'$ (i.e., w ends with w')), and (vi) the end-of-output marker \diamond may not be generated (i.e., $(s \cdot p, w \cdot w', s' \cdot p', w') \in P$ and $w \sqsupseteq \diamond$ imply $w' \sqsupseteq \diamond$). Rules $(s \cdot p, w \cdot w', s' \cdot p', w')$ are written $s \cdot p | w \cdot w' \rightarrow s' \cdot p' | w'$. \square

Intuitively, a rule $s \cdot p | w \cdot w' \rightarrow s' \cdot p' | w'$ is changing the state from p to p' , pops s from the stack, pushes s' to the stack, fixes the output w' , and generates w to the output.

Definition 12. (Parser—ATS Instantiation).

A Parser $M = (N, \Sigma, S, F, P, \diamond)$ instantiates the ATS scheme $(E, C, S, \pi_S, R, c_0, A, O, o_m, o_{um})$ via: (i) $\boxed{E} P$ (ii) $\boxed{C} \mathcal{C}(M) \triangleq N^+ \times \Sigma^* \times \Sigma^*$ where $(s \cdot p, w, f) \in \mathcal{C}(M)$ contains the stack fragment s , the current state p , a history variable w , and the fixed part $f \in \Sigma^{\leq 1}$ which the parser fixed without generating it. (iii) $\boxed{S} \{\bar{p} \in N \mid (s \cdot p, w \cdot w', s' \cdot p', w') \in P \wedge \bar{p} \in \{p, p'\}\}$ (iv) $\boxed{\pi_S} (s \cdot p, w, f) \{p\}$ (v) $\boxed{R} \vdash_M : \mathfrak{X}(P, \mathcal{C}(M))^2$ given by $(e, (s \cdot s_1 \cdot p, w, f)) \vdash_M ((s_1 \cdot p, w_1, s_2 \cdot p', w_2), s \cdot s_2 \cdot p', w', f')$ where (a) $w_1 \sqsubseteq f \vee f \sqsubseteq w_1$, (b) $w' = w \cdot \text{drop}(|f|, \text{delo}(w_1))$,⁸ and finally (c) $f' = w_2 \cdot \text{drop}(|w_1|, f)$. (vi) $\boxed{c_0} (S, \lambda, \lambda)$ (vii) $\boxed{A} \{(s \cdot p, w, f) \mid$

⁵ The here relevant section 6.6 of the monograph Sippu and Soisalon-Soininen (1990) is based primarily on the work of Knuth (1965) which was later extended by Aho and Ullman (1972).

⁶ E.g., $\{(S, A, B), \{a\}, \{(S, A), (S, B), (A, a), (B, a)\}, S\} \notin \text{LR}(1)$.

⁷ An equivalent linear/scheduled definition of Parser is given by Sippu and Soisalon-Soininen (1990).

⁸ Here $\text{delo}(\bar{w})$ removes a potential \diamond from the end of \bar{w} and $\text{drop}(n, \bar{w})$ drops the first n symbols from \bar{w} .

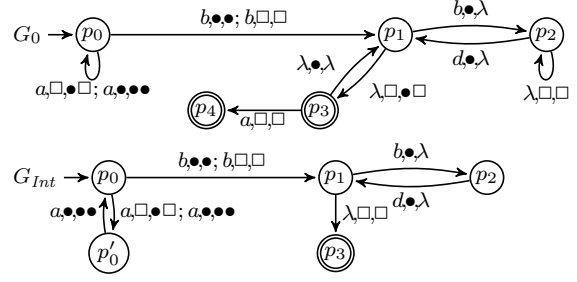


Figure 1. DPDA G_0 and G_{Int} generating $\{a^{2n}b(bd)^n \mid n \in \mathbb{N}\}$.

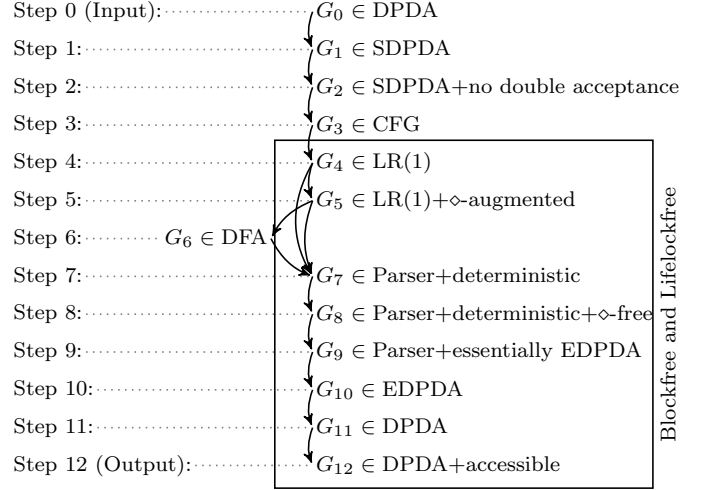


Figure 2. Visualization of the algorithm.

$f \in \{\lambda, \diamond\} \wedge p \in F$ (viii) $\boxed{O} \Sigma^*$ (ix) $\boxed{o_m(s \cdot p, w, f)} \{w\}$ (x) $\boxed{o_{um}(s \cdot p, w, f)} \{w\}$. \square

A Parser is deterministic iff for all reachable configuration all two distinct steps (i) append distinct symbols to the history variable, or (ii) one step adds a symbol to the history variable and the other step completes the output-generation by fixing the end-of-output marker \diamond ⁹.

3. APPROACH

Motivation: For example, the DPDA G_0 in Figure 1 exhibits a lifelock generating the output b reaching p_1, p_3 arbitrarily often, a lifelock (and blocking situations) generating the output abb reaching p_2 arbitrarily often, a non accessible state p_4 (along with the edge leading to it), but no deadlock. Observe that the cause (G_0 does not properly distinguish between an even or odd number of generated a s) is structurally separated from the lifelock at p_2 . Thus, the intuitive solution G_{Int} (see Figure 1) is obtained by splitting the state p_0 and by removing junk. Any formal construction must detect the states with a deadlock, a lifelock, or a blocking situation, determine the cause of that problem, and make a decision on how to fix the problem.

Solution: In Figure 2 we have depicted our approach in the subsequently explained 12 steps. The basic idea is to (Steps 1–3) transform the DPDA G_0 into a CFG G_3 , (Step 4)

⁹ A parser may (depending on the other rules) be deterministic if $(p_1, w \cdot \alpha, \lambda)$ and (p_2, w, \diamond) are successors of the same reachable configuration (p, w, λ) .

obtain an LR(1) grammar G_4 by restricting G_3 to establish operational blockfreeness and absence of lifelocks, (Steps 5–11) transform the LR(1) grammar into a DPDA G_{11} preserving the desired properties, and finally (Step 12) remove all inaccessible states and edges.

Steps 1–4 and 7–12 preserve the marked language. Steps 1–3 and 7–12 preserve the unmarked language while step 4 restricts the unmarked language to the prefix closure of the marked language. Steps 5 and 6 are not meant to preserve the (un)marked language as they are only intermediate results of the translation in Step 7.

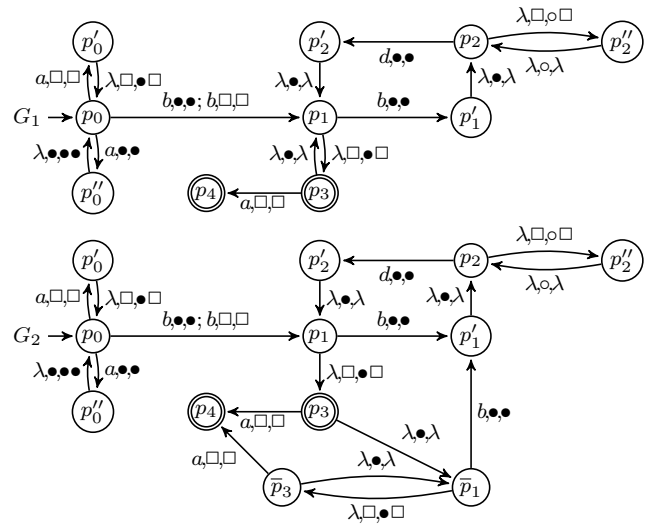
► *Approximating Accessibility*: Throughout the following presentation we omit states and edges which are obviously inaccessible: such states and edges are detected by overapproximating the possible $\leq k$ -length prefixes of stacks in reachable configurations. The k -overapproximation $\mathcal{R} : Q \rightarrow Q \rightarrow 2^{\Gamma^{\leq k}}$ is the least function satisfying the following rules: (i) initial configuration: $k : \square \in \mathcal{R}(p_0, p_0)$, (ii) closure under steps: if $\gamma s \in \mathcal{R}(p, p)$ and $(p, \sigma, \gamma, s', p') \in \delta$ then $k : (s' \cdot s) \in \mathcal{R}(p, p')$ and $k : (s' \cdot s) \in \mathcal{R}(p', p')$, and (iii) transitivity: if $s \in \mathcal{R}(p, p')$ and $s' \in \mathcal{R}(p', p'')$ then $s' \in \mathcal{R}(p, p'')$.¹⁰

For example, in G_0 the state p_4 is obviously inaccessible because the set of all ≤ 1 -length prefixes of stacks of reachable configurations with state p_4 is empty. However, we would obtain λ to be a ≤ 0 -length prefix of a reachable configuration with state p_4 ; i.e., by increasing the parameter for the length of the calculated prefixes a better result may be obtained. For DFA and $k = 0$ the standard DFA-accessibility-operation is obtained. For arbitrary DPDA step 12 alone enforces accessibility.

Applying this approximation implicitly in the running example, we now describe the steps of the algorithm solving the problem stated in Definition 4.

► *Step 1*: We transform the DPDA into a simple DPDA (called SDPDA subsequently) such that every edge is of one of three forms: a generating edge $(p, \alpha, \gamma, \gamma, p')$, a pop edge $(p, \lambda, \gamma, \lambda, p')$, or a push edge $(p, \lambda, \gamma, \gamma', p')$. The operation consists of four steps: (i) split every edge of the form $(p, \alpha, \gamma, s, p')$ into $(p, \alpha, \gamma, \gamma, p'')$ and $(p'', \lambda, \gamma, s, p')$, (ii) split every neutral edge of the form $(p, \lambda, \gamma, \gamma, p')$ into $(p, \lambda, \gamma, \circ, p'')$ and $(p'', \lambda, \circ, \lambda, p')$ for a unique fresh stack symbol $\circ \in \Gamma$, (iii) split every rule of the form $(p, \lambda, \gamma, s\gamma', p')$ with $\gamma \neq \gamma'$ into $(p, \lambda, \gamma, \lambda, p'')$ and $(p'', \lambda, \gamma'', s\gamma'\gamma'', p')$ for every $\gamma'' \in \Gamma$, and (iv) split every rule of the form $(p, \lambda, \gamma, s\gamma, p')$ into $|s|$ steps which push a single symbol of s in each step. Note that the fresh states to be used in each of the four steps contain the edge for which they have been constructed (i.e., p'' in the first step is $(p, \sigma, \gamma, s, p')$). The operation has been adapted from Knuth (1965) by (1) correcting the handling of neutral edges involving the \square symbol (for example, the self loop at p_2 in G_0 would have been handled incorrectly),

¹⁰Without using the transitivity rule we obtain the 0- and 1-overapproximations \mathcal{R}_0 and \mathcal{R}_1 of G_0 (where we omit empty sets):
 $\mathcal{R}_0 = \{p_0 \mapsto \{p_1 \mapsto \{\lambda\}, p_0 \mapsto \{\lambda\}\}, p_1 \mapsto \{p_2 \mapsto \{\lambda\}, p_1 \mapsto \{\lambda\}, p_3 \mapsto \{\lambda\}\}, p_2 \mapsto \{p_2 \mapsto \{\lambda\}, p_1 \mapsto \{\lambda\}\}, p_3 \mapsto \{p_1 \mapsto \{\lambda\}, p_4 \mapsto \{\lambda\}, p_3 \mapsto \{\lambda\}\}, p_4 \mapsto \{p_4 \mapsto \{\lambda\}\}\}$
 $\mathcal{R}_1 = \{p_0 \mapsto \{p_1 \mapsto \{\square, \bullet\}, p_0 \mapsto \{\square, \bullet\}\}, p_1 \mapsto \{p_2 \mapsto \{\lambda\}, p_1 \mapsto \{\lambda, \square, \bullet\}, p_3 \mapsto \{\bullet\}\}, p_2 \mapsto \{p_2 \mapsto \{\lambda, \square\}, p_1 \mapsto \{\lambda\}\}, p_3 \mapsto \{p_1 \mapsto \{\lambda\}, p_3 \mapsto \{\bullet\}\}\}$



G_4 with initial symbol $L_{p_0, \square}$	
1: $L_{p_0, \square} \rightarrow a \cdot L_{p'_0, \square}$	8: $L_{p_0, \square} \rightarrow b \cdot L_{p_1, \square}$
2: $L_{p'_0, \square} \rightarrow L_{p_0, \bullet, p_1} \cdot L_{p_1, \square}$	9: $L_{p_0, \bullet, p_1} \rightarrow a \cdot L_{p'_0, \bullet, p_1}$
3: $L_{p_1, \square} \rightarrow L_{p_3, \bullet}$	10: $L_{p_3, \bullet} \rightarrow \lambda$
4: $L_{p'_0, \bullet, p_1} \rightarrow L_{p_0, \bullet, p_2} \cdot L_{p_2, \bullet, p_1}$	11: $L_{p_0, \bullet, p_2} \rightarrow a \cdot L_{p'_0, \bullet, p_2}$
5: $L_{p_0, \bullet, p_2} \rightarrow b \cdot L_{p_1, \bullet, p_2}$	12: $L_{p_1, \bullet, p_2} \rightarrow b \cdot L_{p'_1, \bullet, p_2}$
6: $L_{p'_1, \bullet, p_2} \rightarrow \lambda$	13: $L_{p'_0, \bullet, p_2} \rightarrow L_{p_0, \bullet, p_1} \cdot L_{p_1, \bullet, p_2}$
7: $L_{p_2, \bullet, p_1} \rightarrow d \cdot L_{p'_2, \bullet, p_1}$	14: $L_{p'_2, \bullet, p_1} \rightarrow \lambda$
renamed G_4 with initial symbol S	
1: $S \rightarrow aA$	2: $A \rightarrow BC$
3: $C \rightarrow D$	4: $E \rightarrow FJ$
5: $F \rightarrow bG$	6: $H \rightarrow \lambda$
7: $J \rightarrow dK$	8: $S \rightarrow bC$
9: $B \rightarrow aE$	10: $D \rightarrow \lambda$
11: $F \rightarrow aI$	12: $G \rightarrow bH$
13: $I \rightarrow BG$	14: $K \rightarrow \lambda$

Figure 3. The simple DPDA G_1 , the simple DPDA G_2 not exhibiting double marking, and the LR(1)-grammar G_4 .

and by (2) logging the involved edges in the fresh states as explained before. For the DPDA G_0 from Figure 1 the SDPDA G_1 in Figure 3 results (up to renaming of the states).

► *Step 2*: We transform the SDPDA G_1 into an SDPDA G_2 such that once the SDPDA G_2 has generated an output, it has to generate another symbol before entering a marking state again. For the example automaton G_1 this means that the lifelock at p_2, p_3 is problematic. We are reusing the construction from Knuth (1965): Every state is duplicated (the duplicated states are neither initial nor marking). Then, the edges are defined such that the automaton G_2 operates on the original states until it reaches a marking state. Once this happens, the automaton either remains in the original states by using a generating edge or it switches to the duplicated states. The automaton remains in the duplicated states until switching to the original states using any generating edge. For the SDPDA G_1 from Figure 3 the SDPDA G_2 in the same figure results. Note, the lifelock in p_1, p_3 has been removed by the cost of another lifelock in \bar{p}_1, \bar{p}_3 generating the same output b .

► *Step 3 & Step 4*: We transform the SDPDA G_2 in step 3 into the CFG G_3 using a construction from Knuth (1965). We restrict the CFG G_3 in step 4 to the LR(1) grammar G_4 (see Figure 3) by removing all productions from G_3 which do not appear in any marking derivation of G_3 . That is, the accessible and coaccessible part is constructed using

SDPDA G_2	LR(1) G_4
(p_0, λ, \square)	$L_{p_0, \square}$
$\vdash_{G_2} (p'_0, a, \square)$	$\vdash_{G_4} a \cdot L_{p'_0, \square}$
$\vdash_{G_2} (p_0, a, \bullet \square)$	$\vdash_{G_4} a \cdot L_{p_0, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p'_0, aa, \bullet \square)$	$\vdash_{G_4} aa \cdot L_{p'_0, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p_0, aa, \bullet \bullet \square)$	$\vdash_{G_4} aa \cdot L_{p_0, \bullet, p_2} \cdot L_{p_2, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p_1, aab, \bullet \bullet \square)$	$\vdash_{G_4} aab \cdot L_{p_1, \bullet, p_2} \cdot L_{p_2, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p'_1, aabb, \bullet \bullet \square)$	$\vdash_{G_4} aabb \cdot L_{p'_1, \bullet, p_2} \cdot L_{p_2, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p_2, aabb, \bullet \square)$	$\vdash_{G_4} aabb \cdot L_{p_2, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p'_2, aabbd, \bullet \square)$	$\vdash_{G_4} aabbd \cdot L_{p'_2, \bullet, p_1} \cdot L_{p_1, \square}$
$\vdash_{G_2} (p_1, aabbd, \square)$	$\vdash_{G_4} aabbd \cdot L_{p_1, \square}$
$\vdash_{G_2} (p_3, aabbd, \bullet \square)$	$\vdash_{G_4} aabbd \cdot L_{p_3, \bullet}$
	$\vdash_{G_4} aabbd$

Figure 4. Corresponding initial derivations of the SDPDA G_2 and the LR(1) grammar G_4 .

a fixed-point algorithm in each case. For the accessible part: the *accessible* nonterminals are the least set of nonterminals \mathcal{A} such that the initial nonterminal is contained in \mathcal{A} and for any production $A \rightarrow v$: if $A \in \mathcal{A}$, then the nonterminals of v are contained in \mathcal{A} . For the coaccessible part: the *coaccessible* nonterminals are the least set of nonterminals \mathcal{A} such that for any production $A \rightarrow v$: if the nonterminals of v are contained in \mathcal{A} then $A \in \mathcal{A}$. The equivalence of G_2 and G_4 w.r.t. the marked language can best be understood by comparing the derivations in Figure 4. The following three properties explain the correctness of the construction: (i) The nonterminals of the form $L_{p,A}$ (for example $L_{p_1, \square}$) guarantee a marking derivation of the SDPDA starting in p not modifying the stack starting with A . (ii) The nonterminals of the form $L_{p,A,p'}$ (for example L_{p_0, \bullet, p_2}) guarantee a derivation of the SDPDA starting in p not modifying the stack starting with A and reaching a configuration in which the A is removed and the state p' is reached. (iii) For any configuration $(p_1, w, \gamma_1 \dots \gamma_n)$ there are $p_2 \dots p_n$ such that $(p, w, \gamma_1 \dots \gamma_n)$ is reachable by G_2 iff $w \cdot L_{p_1, \gamma_1, p_2} \dots L_{p_{n-1}, \gamma_{n-1}, p_n} L_{p_n, \gamma_n}$ is reachable by G_4 .

Once step 4 has been completed, for the given DPDA a marked language equivalent CFG has been constructed which is lifelockfree, accessible, and operational blockfree (and by that deadlockfree).

► *Step 5 & Step 6 & Step 7:* In these steps we are following, with some modifications, the constructions in Sippu and Soisalon-Soininen (1990).

In step 5 we are constructing the \diamond -augmented version G_5 of G_4 : A new initial nonterminal S' and the production $S' \rightarrow \diamond S \diamond$ are added where S is the old nonterminal. This modification allows for a simpler construction procedure of the LR(1)-machine and the LR(1)-parser in steps 6 and 7.

In step 6 we are constructing the LR(1)-machine G_6 (depicted in Figure 5) for the LR(1)-grammar G_5 . The output alphabet of the DFA G_6 is the union of the output alphabet and the nonterminals of G_5 . The steps of the parser (between two states p, p' of G_6) will depend on the elements of p : these elements are called items which are formally four-tuples containing a production with a marker splitting the right hand side of the production and a lookahead symbol. The DFA G_6 has two kinds of edges: the edges labeled with an output symbol α represent the action where the parser generates α , the edges labeled

Reduce Rules	Shift Rules
1-3-5 $\diamond \rightarrow$ 1-6 \diamond	1-2-7 $\diamond \rightarrow$ 1-6 \diamond
2-8-10 $\diamond \rightarrow$ 2-7 \diamond	2-9-12 $\diamond \rightarrow$ 2-8 \diamond
3-4 $\diamond \rightarrow$ 3-5 \diamond	3 $\diamond \rightarrow$ 3-4 \diamond
8-4 $\diamond \rightarrow$ 8-10 \diamond	8 $\diamond \rightarrow$ 8-4 \diamond
9-13-16 $\diamond \rightarrow$ 9-12 \diamond	13-17-18 $\diamond \rightarrow$ 13-16 \diamond
17 $\diamond \rightarrow$ 17-18 \diamond	9-14-19 $d \rightarrow$ 9-13 d
9-15-22 $d \rightarrow$ 9-13 d	14-26-27 $d \rightarrow$ 14-19 d
15-23-25 $d \rightarrow$ 15-22 d	15-24-28 $b \rightarrow$ 15-23 b
23-26-27 $d \rightarrow$ 23-25 d	24-15-22 $d \rightarrow$ 24-29 d
24-29-30 $b \rightarrow$ 24-28 b	26 $d \rightarrow$ 26-27 d
24-14-19 $d \rightarrow$ 24-29 d	24 $b \rightarrow$ 24-14
29-31-32 $b \rightarrow$ 29-30 b	31 $b \rightarrow$ 31-32 b
	29 $d \rightarrow$ 29-31

Figure 6. The rules of the LR(1)-parser G_7 with initial state 1 and marking set $\{6\}$.

with nonterminals are required for the actions where the parser concludes (based on its stack and the lookahead of the items) that it has generated a word derivable by a nonterminal.

Every edge (p, κ, p') in G_6 satisfies that p' is the least set satisfying the following conditions: p' contains all items of p where the marker \succ has been shifted over κ . Furthermore, if an item of the form $[A \rightarrow v \succ B \cdot v', \sigma]$ is obtained, then the so-called “first”-symbols σ' are determined¹¹ for which there is a w satisfying $v' \cdot \sigma \vdash_{G_5}^* w$ with $\sigma' = 1:w$ and for all such (possibly empty) σ' and all productions of the form $B \rightarrow v''$, the item $[B \rightarrow v'' \succ \sigma', \sigma']$ is contained in p' ¹².

For example (in Figure 5), the a -successor of state 9 is state 15: $[F \rightarrow a \succ I, d] \in 15$ is the result of the shifting of the \succ over the a in the item $[F \rightarrow \succ aI, d] \in 9$; $[I \rightarrow \succ BG, d] \in 15$ because $[F \rightarrow a \succ I, d] \in 15$ and d is (trivially) derivable to d ; $[B \rightarrow \succ aE, b] \in 15$ because $[I \rightarrow \succ BG, d] \in 15$ and Gd is derivable to bd .

In step 7 we are constructing the LR(1)-parser G_7 (depicted in Figure 6) for G_5 and G_6 . The parser consists of shift rules (generating a symbol and changing the stack and state) and reduce rules (which only modify the stack and state). The shift rules are obtained from the LR(1)-machine by selecting the edges in G_6 which are labeled with an output symbol: an edge (p, α, p') would result in the shift rule $p|\alpha \rightarrow p \cdot p'|\lambda$ (e.g., the edge $(1, a, 2)$ results in the rule $1|a \rightarrow 1 \cdot 2|\lambda$). The reduce rules are constructed for every item of the form $[A \rightarrow \succ v, \alpha] \in p$ (i.e., the marker \succ is at the beginning of the right hand side): let \tilde{p} (by construction \tilde{p} is also a word over the stack alphabet of G_7) be the sequence of states visited by generating v starting in p in G_6 and let p' be the state reached by generating A in p in G_6 . Then the reduce rule $p \cdot \tilde{p}|\lambda \rightarrow p \cdot p'|\lambda$ is added to the parser (e.g., the item $[J \rightarrow \succ dK, b] \in 29$ results in the rule $29 \cdot 31 \cdot 32|b \rightarrow 29 \cdot 30|b$).

Remark 2. According to Sippu and Soisalon-Soininen (1990), the parser G_7 is a *correct prefix parser*. However, that is a too weak assertion: their definition of the unmarked language considers a symbol the parser has fixed but not generated not to be part of the generated

¹¹ While in Sippu and Soisalon-Soininen (1990) no effective algorithm is presented for this operation we have been able to verify such a construction.

¹² The state with no items has been removed from the visualization in Figure 5.

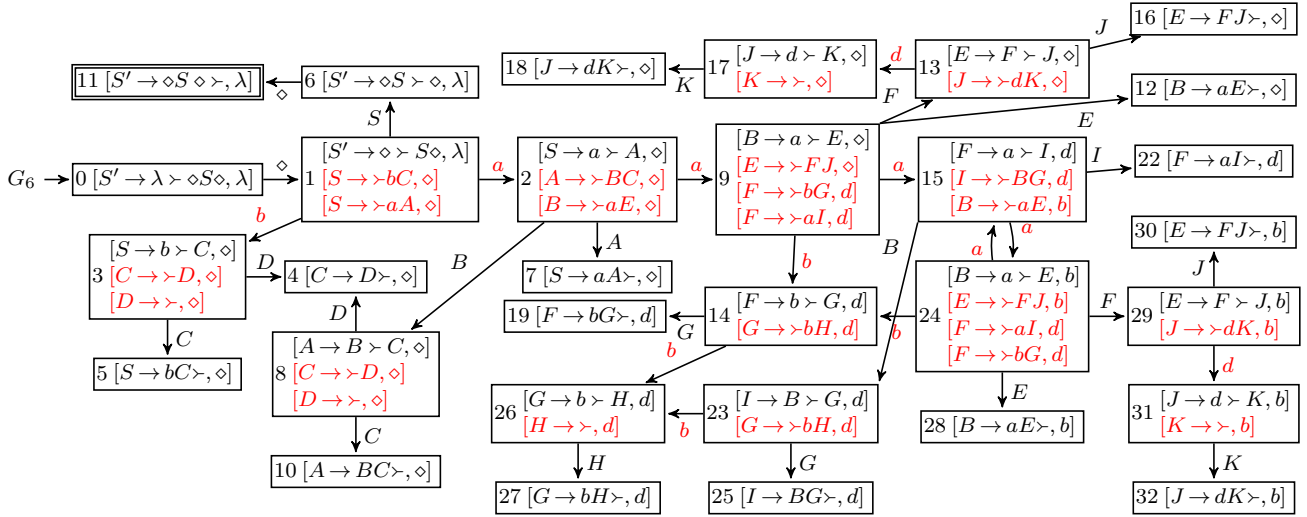


Figure 5. The LR(1)-machine G_6 . Edges generating terminals (relevant for shift-rules) and items with marker \succ at the beginning of the right hand side (relevant for reduce rules) are printed in red.

Reduce Rules	Shift Rules	Reduce Rules
9-14-19 d→ 9-13 d	1 a→ 1-2	9-14-(19, λ) d→ 9-(13, d) λ
9-15-22 d→ 9-13 d	1 b→ 1-3	24-14-(19, λ) d→24-(29, d) λ
15-23-25 d→15-22 d	2 a→ 2-9	9-15-(22, λ) d→ 9-(13, d) λ
14-26-27 d→14-19 d	9 a→ 9-15	24-15-(22, λ) d→24-(29, d) λ
15-24-28 b→15-23 b	9 b→ 9-14	15-23-(25, λ) d→15-(22, d) λ
31 b→31-32 b	13 d→13-17	(26, λ) d→26-(27, d) λ
	14 b→14-26	14-26-(27, λ) d→14-(19, d) λ
	15 a→15-24	23-26-(27, λ) d→23-(25, d) λ
	23 b→23-26	23-26-(27, d) λ→23-(25, d) λ
	24 a→24-15	15-24-(28, λ) b→15-(23, b) λ
	24 b→24-14	15-24-(28, b) λ→15-(23, b) λ
	29 d→29-31	24-29-(30, λ) b→24-(28, b) λ
		(31, λ) b→31-(32, b) λ
		(31, b) λ→31-(32, b) λ
		29-31-(32, λ) b→29-(30, b) λ
		29-31-(32, b) λ→29-(30, b) λ

Figure 7. The rules of the LR(1)-parser G_8 with initial state 1 and marking set $\{3, 17\}$ (the nonterminals $\{4, 5, 7, 8, 10, 12, 16, 18\}$ are no longer reachable)

unmarked word. Since the mode of operation we are interested (control of (embedded) discrete event systems), we had to find new proofs to verify that our stronger condition is also satisfied by the generated parser G_7 . \square

► **Step 8:** Since DPDA are not capable of terminating the generation by fixing an end-of-output marker, we are modifying the parser G_7 by removing all rules involving the end-of-output marker \diamond and by changing the set of marking states such that G_8 (depicted in Figure 7) marks in $(s \cdot p, w, f)$ iff some edge $s' \cdot p | \diamond \rightarrow s'' | \diamond$ has been removed. While it is not mentioned in Sippu and Soisalon-Soininen (1990), we discovered that this drastic removal of rules preserves the (un)marked language because the parser reaches a configuration in which such an edge is enabled if and only if the stack can be entirely reduced by subsequently executed reduce rules. This optimization also speeds up the parsing process using the presented construction in any other context (e.g., parsing of programming languages for which it has originally been designed).

► **Step 9:** Since DPDA are not capable of fixing output symbols without generating them, we add the fixed output component of a configuration into the state of the configuration. For every shift rule of the form $p | \alpha \rightarrow p' \cdot p' | \lambda$ the rules $(p, \lambda) | \alpha \rightarrow p' \cdot (p', \lambda) | \lambda$ and $(p, \alpha) | \lambda \rightarrow p' \cdot (p', \lambda) | \lambda$ are used. For every reduce rule of the form $s \cdot p | \alpha \rightarrow s' \cdot p' | \alpha$ the

Shift Rules
(1, λ) a→ 1-(2, λ) λ
(1, λ) b→ 1-(3, λ) λ
(2, λ) a→ 2-(9, λ) λ
(9, λ) a→ 9-(15, λ) λ
(9, λ) b→ 9-(14, λ) λ
(13, λ) d→13-(17, λ) λ
(14, λ) b→14-(26, λ) λ
(15, λ) a→15-(24, λ) λ
(23, λ) b→23-(26, λ) λ
(24, λ) a→24-(15, λ) λ
(24, λ) b→24-(14, λ) λ
(29, λ) d→29-(31, λ) λ
(1, a) λ→ 1-(2, λ) λ
(1, b) λ→ 1-(3, λ) λ
(2, a) λ→ 2-(9, λ) λ
(9, a) λ→ 9-(15, λ) λ
(9, b) λ→ 9-(14, λ) λ
(13, d) λ→13-(17, λ) λ
(14, b) λ→14-(26, λ) λ
(15, a) λ→15-(24, λ) λ
(23, b) λ→23-(26, λ) λ
(24, a) λ→24-(15, λ) λ
(24, b) λ→24-(14, λ) λ
(29, d) λ→29-(31, λ) λ

Figure 8. The rules of the LR(1)-parser G_9 with initial state 1 and marking set $\{(3, \lambda), (17, \lambda)\}$.

rules $s \cdot (p, \lambda) | \alpha \rightarrow s' \cdot (p', \alpha) | \lambda$ and $s \cdot (p, \alpha) | \lambda \rightarrow s' \cdot (p', \alpha) | \lambda$ are used. The resulting parser G_9 is depicted in Figure 8.

It is then possible to verify, that all reachable configurations of the resulting parser G_9 have an empty fixed output component. We call the parser G_9 essentially EDPDA because it uses none of the extra capabilities of the parser formalism.

► **Step 10:** The essentially EDPDA parser G_9 can be translated into the EDPDA G_{10} (depicted in Figure 9) by using for every rule of the form $s \cdot p | \sigma \rightarrow s' \cdot p' | \lambda$ the edge $(p, \sigma, s^{-1}, s'^{-1}, p')$. Marking and initial states of G_{10} are taken from G_9 .

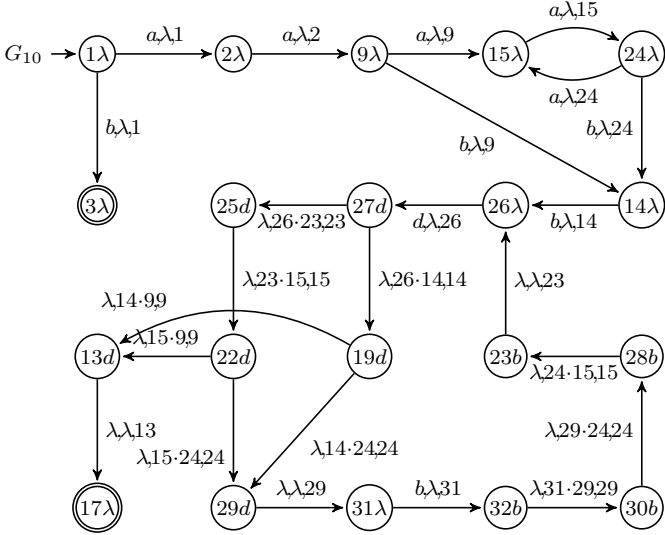


Figure 9. The resulting EDPDA G_{10} where obviously unreachable states have been removed.

► **Step 11:** Since DPDA are not capable of popping strictly *more* than one symbol from the stack, we split such edges into multiple edges to obtain the DPDA G_{11} . To preserve determinism, the splitting of edges with the same source entails the merging of partially identical edges until the recursive split identifies their distinctness. For example, the edges $(p, \sigma, s \cdot s', s_1, p_1)$ and $(p, \sigma, s \cdot s'', s_2, p_2)$ share a common prefix s on the popping component.

Since DPDA are not capable of popping strictly *less* than one symbol from the stack, we modify the automaton by replacing any edge $(p, \sigma, \lambda, s, p')$ with $(p, \sigma, \gamma, s \cdot \gamma, p)$ for any γ of the stack alphabet of G_{10} . For soundness, recall that the stack-bottom-marker can never be removed from the stack.

► **Step 12:** Finally, accessibility of states and edges can be enforced by reusing the presented steps 1–4. For a DPDA we are executing steps 1–4. From the productions obtained by step 4 we can determine by executing the steps 1–3 backwards (which are by our construction injective in the sense that for each constructed production/edge x a unique edge e can be determined for which x has been constructed). Using this backwards computation, we are able to determine the accessible edges of a DPDA. The accessible states are the sources and edges of any of the accessible edges. The inaccessible states and edges are then removed to obtain the DPDA G_{12} from Figure 10.

We are not aware of comparable constructions ensuring accessibility of DPDA, however, using the decidability of emptiness from Hopcroft and Ullman (1979) it is possible to test a single (and by that every) edge for accessibility; this approach has been used in Griffin (2006). Our approach is superior as we are executing a single test on all edges simultaneously.

► **Verification:** The soundness of the presented algorithm (w.r.t. the problem Definition 4) has been verified in the interactive theorem prover Isabelle/HOL (Paulson et al., 2011) apart from the following steps for which only pen-and-paper proofs exist yet and which are to be completed

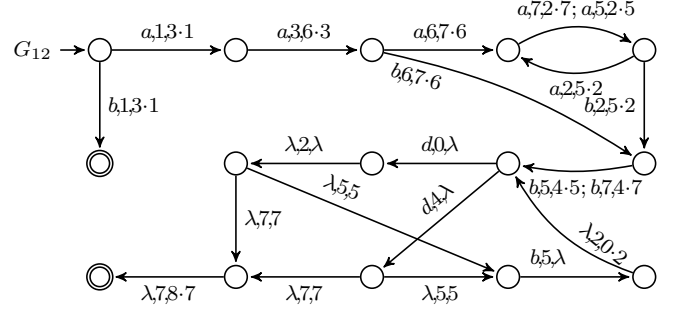


Figure 10. The resulting DPDA G_{12} .

in Isabelle/HOL in the near future: (i) the CFG obtained in step 4 is an LR(1) grammar (satisfied according to Knuth (1965)), (ii) the Parser obtained in step 7 is deterministic if G_5 is an LR(1) grammar (satisfied according to Sippu and Soisalon-Soininen (1990)), (iii) step 11, and (iv) step 12. From these tasks however, only the first appears to be complicated.

► **Testing:** The presented algorithm has been implemented in Java for rapid prototyping and in C++ as a plugin to the libFAUDES (2006-2013) tool. The implementations have been used successfully for many examples including the running example of this paper.

► **Optimizations:** The algorithm can be optimized in different ways. (i) The runtime of the algorithm depends primarily on the steps 3 and 4 because G_3 would have an enormous amount of productions. We can greatly restrict the set of productions to be generated by exploiting the structure of the input DPDA using the reachability overapproximation presented on page 4. (ii) Furthermore, steps 3 and 4 can be merged such that only productions are generated which are coaccessible. This alternative trades runtime for space-requirements (the size of G_4 is usually not much greater than G_1 but the runtime is increased by the length of the longest derivation necessary in G_2 to reach all states). (iii) Another optimization merges adjacent edges in EDPDA which are intermediate results. This optimization decreases the runtime of the subsequently executed operations. The formal definition and verification in Isabelle/HOL of such intermediate operations is left for future work.

4. CONCLUSION

The algorithm presented in this paper optimizes the behavior of a DPDA whilst preserving its marked language by first translating the DPDA into another model (LR(1) grammars) in which the desired properties can be enforced using simple constructions and by translating the obtained solution back into DPDA while preserving the desired properties.

The algorithm guarantees accessibility (every state and every edge is required for some marking derivation), life-lockfreeness (there is no initial derivation executing infinitely many steps without generating an output symbol), deadlockfreeness (non-extendable initial derivations are ending in marking states), and finally the operational blockfreeness (every initial derivation can be extended into a marking derivation).

The operational blockfreeness is sufficient to conclude that the unmarked language is the prefix closure of the marked language of the resulting DPDA.

The algorithm does not minimize the size of the automaton, in fact, the size of the resulting DPDA is usually increased and is growing according to Geller et al. (1975) in some cases exponentially.

The algorithm presented here is a crucial part of the presented solution of the supervisory control problem for DFA plants and DPDA specifications which is reduced (in the companion paper by Schneider, Schmuck, Raisch, and Nestmann (2014)) to the effective implementability of ensuring blockfreeness (solved in this paper) and ensuring controllability (solved in the companion paper by Schmuck, Schneider, Raisch, and Nestmann (2014)).

5. FUTURE WORK

► *Petri nets*: Since the problem of establishing blockfreeness is unsolvable for standard Petri net classes (Giua and DiCesare, 1994, 1995), we intend to determine Petri net classes \mathcal{P} that can be translated (preserving the marked language) into a DPDA G such that the DPDA generated by our algorithm G' can be translated back into a Petri net from \mathcal{P} to solve the problem for such a Petri net class.

► *Visibly Pushdown Tree Automata (VPTA)*: VPTA introduced by Chabin and Réty (2007) are the greatest known subclass of DPDA which are closed under intersection. For the context of the Supervisory Control Theory we intend to determine an algorithm which solves the problem from Definition 4 for VPTA because (i) plant and controller can then be generated by VPTA, while this decreases the expressiveness for the controller language it also increases the expressiveness for the plant language, and (ii) the closed loop is again a VPTA, which allows for the iterative restriction of a plant language by horizontal composition of controllers. The algorithm presented here may be reusable: the output of the algorithm, when executed on a VPTA, may be (convertible) into a VPTA. Therefore, when using VPTA for plants, specifications, and controllers, the supervisory controller synthesis can be extended to yet another domain.

► *Nondeterminism*: For the context of the Supervisory Control Theory there is no reason to restrict oneself to deterministic controllers. However, for these systems the desired property of operational blockfreeness is not guaranteed for language blockfree controllers. Therefore, when extending the domain of the algorithm to PDA the proofs will become more complex as the preservation of marked and unmarked language is no longer sufficient for the preservation of the operational blockfreeness as discussed in Schneider et al. (2014).

REFERENCES

- Aho, A.V. and Ullman, J.D. (1972). *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Chabin, J. and Réty, P. (2007). Visibly pushdown languages and term rewriting. In B. Konev and F. Wolter (eds.), *FroCoS*, volume 4720 of *Lecture Notes in Computer Science*, 252–266. Springer.
- Geller, M.M., III, H.B.H., Szymanski, T.G., and Ullman, J.D. (1975). Economy of descriptions by parsers, dpda's, and pda's. In *FOCS*, 122–127. IEEE Computer Society.
- Ginsburg, S. and Greibach, S.A. (1966). Deterministic context free languages. *Information and Control*, 9(6), 620–648.
- Giua, A. and DiCesare, F. (1994). Blocking and controllability of petri nets in supervisory control. *IEEE Transactions on Automatic Control*, 39(4), 818–823. doi: 10.1109/9.286260.
- Giua, A. and DiCesare, F. (1995). Decidability and closure properties of weak petri net languages in supervisory control. *IEEE Transactions on Automatic Control*, 40(5), 906–910. doi:10.1109/9.384227.
- Griffin, C. (2006). A note on deciding controllability in pushdown systems. *IEEE Transactions on Automatic Control*, 51(2), 334 – 337.
- Hopcroft, J.E. and Ullman, J.D. (1979). *Introduction to Automata Theory, languages and computation*. Addison-Wesley Publishing company.
- Knuth, D.E. (1965). On the translation of languages from left to right. *Information and Control*, 8(6), 607–639.
- libFAUDES (2006-2013). Software library for discrete event systems. URL <http://www.rt.eei.uni-erlangen.de/FGdes/faudes>.
- Paulson, L., Nipkow, T., and Wenzel, M. (2011). Isabelle/HOL. URL <http://isabelle.in.tum.de>.
- Schmuck, A.-K., Schneider, S., Raisch, J., and Nestmann, U. (2014). Extending supervisory controller synthesis to deterministic pushdown automata—enforcing controllability least restrictively. *WODES'14*.
- Schneider, S. and Schmuck, A.-K. (2013). Supervisory controller synthesis for deterministic pushdown automata specifications. Technical report, Technical University of Berlin, URL <http://www.tu-berlin.de/?25631>.
- Schneider, S., Schmuck, A.-K., Raisch, J., and Nestmann, U. (2014). Reducing an operational supervisory control problem by decomposition for deterministic pushdown automata. *WODES'14*.
- Sippu, S. and Soisalon-Soininen, E. (1990). *Parsing Theory*, volume II: LR(k) and LL(k) Parsing of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag.